Parte TRÊS

Expressões Lambda

Capítulo OITO

Interfaces Funcionais

Objetivos do Exame
Criar e usar expressões Lambda.

Simplificando as coisas

Suponha que temos um programa que possui uma lista de carros e precisamos buscar todos os carros compactos.

Podemos ter algo assim para fazer o trabalho:

```java
List<Car> findCompactCars(List<Car> cars) {
    List<Car> compactCars = new ArrayList<Car>();
    for(Car car : cars) {
        if(car.getType().equals(CarTypes.COMPACT)) {
            compactCars.add(car);
        }
    }
    return compactCars;
}
```

Fácil. Mas no dia seguinte, os usuários percebem que também precisam buscar por carros que custem mais de 20.000 dólares.

Então criamos algo assim:

```java
List<Car> findCompactCars(List<Car> cars) {
    List<Car> twentyKCars = new ArrayList<Car>();
    for(Car car : cars) {
        if(car.getCostUSD() > 20000) {
            twentyKCars.add(car);
        }
    }
    return twentyKCars;
}
```

Agora olhe para o código. É praticamente o mesmo. As únicas diferenças são as linhas:

```java
car.getType().equals(CarTypes.COMPACT)
```

E:

```java
car.getCostUSD() > 20000
```

E se precisarmos filtrar por outra condição no futuro?

É errado ter código duplicado ou copiado e colado, não é muito flexível para mudanças e propenso a erros.

A Abordagem Orientada a Objetos

Como o Java é orientado a objetos, vamos aproveitar isso resolvendo o problema usando um padrão de projeto popular, Strategy (Estratégia).

Esse padrão faz exatamente o que precisamos: encapsula o comportamento que varia (algoritmo) e torna todos esses comportamentos intercambiáveis.

A forma recomendada de implementar o padrão Strategy em Java é com uma interface, para que possamos criar implementações para cada algoritmo (neste caso, as condições a serem testadas).

Então vamos codificar uma interface que pode conter a condição de busca que varia, por exemplo:

java Copiar Editar

```java
interface Searchable {
    boolean test(Car car);
}
```

E colocamos o código que varia em implementações dessa interface:

java Copiar Editar

```java
class CompactCarSearch implements Searchable {
    public boolean test(Car car) {
        return car.getType().equals(CarTypes.COMPACT);
    }
}
```

E:

java Copiar Editar

```java
class TwentyKCarSearch implements Searchable {
    public boolean test(Car car) {
        return car.getCostUSD() > 20000;
    }
}
```

Dessa forma, podemos tornar o método que faz a busca real um pouco mais geral:

java Copiar Editar

```java
List<Car> findCars(List<Car> cars, Searchable s) {
    List<Car> searchedCars = new ArrayList<Car>();
    for(Car car : cars) {
        if(s.test(car)) {
            searchedCars.add(car);
        }
    }
    return searchedCars;
}
```

E chamar a função desta forma:

java Copiar Editar

```java
List<Car> compactCars = findCars(cars, new CompactCarSearch());
List<Car> twentyKCars = findCars(cars, new TwentyKCarSearch());
```

Não temos mais código duplicado.

Se o usuário quiser outra forma de buscar por um carro, em vez de copiar e colar um método de busca, agora apenas implementamos outra classe com uma condição.

Agora somos mais flexíveis para mudar.

Mas ainda é uma solução COMPLEXA. Em vez de dois métodos, temos duas classes e uma interface.

E se transformássemos as implementações em classes anônimas? Por exemplo:

```java
List<Car> compactCars = findCars(cars, new Searchable() {
    public boolean test(Car car) {
        return car.getType().equals(CarTypes.COMPACT);
    }
});
```

Um pouco melhor, podemos até atribuir a classe a uma variável, se quisermos usá-la em outras partes do programa:

```java
Searchable compactCarSearch = new Searchable() {
    public boolean test(Car car) {
        return car.getType().equals(CarTypes.COMPACT);
    }
};
List<Car> compactCars = findCars(cars, compactCarSearch);
```

Nada mal, mas não seria ótimo apenas passar a condição ao método? Como isto, por exemplo:

```java
// Nem tente, isso não funcionará
List<Car> compactCars = findCars(cars,
            car.getType().equals(CarTypes.COMPACT));
```

Um novo tipo de valor

Basicamente, em Java, trabalhamos com dois tipos de valor: valores primitivos e referências para objetos.

Ambos podem ser usados como argumentos de um método:

```java
method1(3);
method2(new Object());
```

Então, se quisermos passar um trecho de código para um método, temos que encapsulá-lo em um objeto (como no exemplo anterior).

Mas no Java 8, podemos passar esse trecho de código diretamente através do uso de uma **expressão lambda**.

Coincidentemente, para usar uma expressão lambda em vez de um objeto no exemplo anterior, temos que começar com a mesma interface:

```java
interface Searchable {
    boolean test(Car car);
}
```

Uma interface funcional

O ponto de partida para aprender sobre expressões lambda é aprender sobre interfaces funcionais.

Uma **interface funcional** é qualquer interface que tenha **exatamente UM MÉTODO ABSTRATO**.

Essa é uma definição complicada. Muitas pessoas pensam que apenas ter um método torna uma interface funcional (já que métodos de interface são abstratos por padrão), mas não é bem assim.

Veja, por exemplo, métodos default. Como métodos default possuem implementação, eles **não são abstratos**. Portanto, interfaces como as seguintes são consideradas funcionais:

```java
interface A {
    default int defaultMethod() {
        return 0;
    }
    void method();
}
interface B {
    default int defaultMethod() {
        return 0;
    }
    default int anotherDefaultMethod() {
        return 0;
    }
    void method();
}
```

Se uma interface declara um método abstrato com a assinatura de um dos métodos de java.lang.Object, ele **não conta** para a contagem de métodos abstratos da interface funcional, já que qualquer implementação da interface terá uma implementação do método (pois todas as classes estendem java.lang.Object).

Portanto, uma interface como a seguinte é considerada funcional:

```java
interface A {
    boolean equals(Object o);
    int hashCode();
    String toString();
    void method();
}
```

Um cenário mais confuso é quando uma interface herda um método que é equivalente, mas não idêntico a outro:

```java
interface A {
    void method(List<Double> l);
}
interface B extends A {
    void method(List l);
}
```

Nesse caso, o método é o mesmo, então é considerado como um único método. A classe que implementar B terá que implementar o método que pode legalmente sobrescrever todos os métodos abstratos:

```java
void method(List l);
```

Aliás, uma interface vazia **não** é considerada funcional.

A chave aqui é ter EXATAMENTE UM MÉTODO ABSTRATO; é por isso que essas interfaces também são chamadas de **interfaces de método abstrato único (SAM - Single Abstract Method)**.

Para facilitar, o Java 8 também introduziu a anotação @FunctionalInterface, que gera um erro de compilação quando a interface anotada não é uma interface funcional válida.

```java
// Isto não compilará
@FunctionalInterface interface A {
    void m(int i);
    void m(long l);
}
```

Interfaces Java que possuem apenas um método declarado em sua definição agora são anotadas como interfaces funcionais. Alguns exemplos são:

- java.lang.Runnable
- java.util.Comparator
- java.util.concurrent.Callable
- java.awt.event.ActionListener

Mas lembre-se, essa anotação serve apenas para ajudar; tê-la não torna uma interface funcional.

No próximo capítulo, veremos como interfaces funcionais e expressões lambda estão relacionadas.

Pontos-chave

- Uma interface funcional é qualquer interface que tenha exatamente um método abstrato.
- Como métodos default têm implementação, eles não são abstratos, então uma interface funcional pode ter qualquer número deles.
- Se uma interface declara um método abstrato com a assinatura de um dos métodos de java.lang.Object, ele não conta para a contagem.
- Uma interface funcional é válida mesmo que herde um método equivalente, mas não idêntico a outro.
- Uma interface vazia não é considerada funcional.

- Uma interface funcional é válida mesmo sem a anotação @FunctionalInterface.
- Interfaces funcionais são a base das expressões lambda.

Autoavaliação

1. Dado:

```java
interface A {
    default int m1() {
        return 0;
    }
}
@FunctionalInterface
public interface B extends A {
    static String m() {
        return "static";
    }
}
```

Quais das seguintes afirmações são verdadeiras?
A. A compilação falha
B. Compila com sucesso
C. Compila somente se a interface B declarar um método default
D. Uma exceção ocorre em tempo de execução se essa interface for implementada por uma classe

---

2. Dado:

```java
@FunctionalInterface interface C {
    int m(int i);
    long m(long i);
}
```

Quais das seguintes afirmações são verdadeiras?
A. Este código compila com sucesso
B. Se removermos a anotação, este código compilará
C. Se removermos um método, este código compilará
D. A anotação @FunctionalInterface torna esta interface funcional

---

3. Dado:

```java
public interface D {
    int sum(int a, int b);
    default int subtract(int a, int b) {
        return a - b;
    }
}
```

Quais das seguintes afirmações são verdadeiras?

A. Este código compila com sucesso
B. Este código não compila
C. Este é um exemplo de uma interface funcional
D. Remover o método sum tornaria esta interface funcional

---

4. Quais das seguintes interfaces da API Java podem ser consideradas funcionais?

A. java.util.concurrent.Callable
B. java.util.Map
C. java.util.Iterator
D. java.lang.Comparable

---

5. Dado:

```java
public interface E {
    static int sum(int a, int b) {
        return a + b;
    }
}
```

Quais das seguintes afirmações são verdadeiras?

A. Este código não compila
B. Este código compila com sucesso
C. Este é um exemplo de interface funcional
D. Adicionar a anotação @FunctionalInterface tornaria esta interface funcional